



ACTiCLOUD: ACTivating resource efficiency and large databases in the CLOUD

Project No: 732366

H2020-ICT-2016-1

D3.5: ACTiCLOUD-enabled system libraries v2.0

Due date of deliverable:	M32 (2019/08/31)
Actual submission date:	M33 (2019/09/13)

Executive summary

This deliverable is tightly coupled with deliverables D1.1: “ACTiCLOUD Requirements and base architecture” and D1.2: “ACTiCLOUD architecture”. It is an extension of deliverable D3.1: “ACTiCLOUD-enabled system libraries v1.0” and strongly linked to the components that are put together in the ACTiCLOUD prototype.

The major achievements of the ACTiCLOUD libraries are the improved memory-allocator NC-ALLOC, and the other NUMA aware libraries (NC-LAPACK, NC-MPI, NC-BLACS, and NC-BTL) that make ACTiCLOUD instances available to Big Data, Analytics and HPC applications that need a lot of memory and low latency. Applications using NC-ALLOC, NC-LAPACK, NC-MPI, NC-BLACS, and NC-BTL show good scaling in benchmarks.

Deliverable D3.1: “ACTiCLOUD-enabled system libraries” showed that the toolbox is ready for scaling the applications and databases in ACTiCLOUD to larger memory and lower latencies than used before in a general-purpose cloud solution.

While evaluating NC-ALLOC we have found that given the current memory allocation pattern of MonetDB and Neo4j in-memory databases, the ACTiCLOUD system libraries will not significantly improve their performance. MonetDB and Neo4j allocate one big chunk of memory that they keep using during the whole execution. This prevents fine-grained allocating/freeing smaller chunks of memory with potentially better locality characteristics via NC-ALLOC. However, the unmatched scaling of applications that require frequent allocation/deallocation of memory by NC-ALLOC has been proved with other applications that perform a lot of allocation and deallocation operations such as Astrophysics code from University of Keele and synthetic benchmarks. The principles and part of the NC-ALLOC code are being considered for integration

in the HyperScale JVM being developed by UNIMAN.

During the development of ACTiCLOUD it became apparent that there is great interest in performance counters for Numascale node controllers (from Numascale, UMU, ICCS and UNIMAN). As one of the purposes for the work on ACTiCLOUD system libraries is to scale up workloads, a performance counter framework has been implemented and adapted in order to support code optimization on big scale-up systems like the Numascale shared memory systems in ACTiCLOUD.

List of authors:

Authors	Affiliation
Atle Vesterkjær	NSCALE
Stefan Olbrich	NSCALE
Daniel J Blueman	NSCALE

Dissemination Level	X	PU (Public)
		PP (Restricted to other programme participants)
		RE (Restricted to a group specified by the consortium)
		CO (Confidential, only for members of the consortium)
		Where restricted, access granted to:
Nature		R (Report)
		P (Prototype)
		D (Demonstrator)
	X	O (Other)

Review Status		Draft
		WP Leader accepted
		QA approved
	X	Coordinator accepted

Revision History:

Version	Author(s) (Affiliation)	Notes
0.1	Atle Vesterkjær (NSCALE)	TOC and initial content inserted
0.9	Atle Vesterkjær (NSCALE)	First version ready for review
1.0	Daniel J Blueman (NSCALE)	Review; content and style fixes; complete performance counter section
1.2	Atle Vesterkjær (NSCALE)	Revisions after internal review
1.3	Vasileios Karakostas (ICCS)	Submitted version

ACTiCLOUD Consortium:

Participant No	Participant organisation name	Short name	Country
1 (Coordinator)	Institute of Communication and Computer Systems	ICCS	Greece
2	Numascale AS	NSCALE	Norway
3	Kaleao Limited	KALEAO	UK
4	OnApp Limited	ONAPP	Gibraltar
5	University of Manchester	UNIMAN	UK
6	MonetDB Solutions BV	MDBS	Netherlands
7	Neo Technology	NEO	Sweden
8	UMEA University	UMU	Sweden



NUMASCALE



Confidentiality:

This document contains proprietary and confidential material of certain ACTiCLOUD contractors, and may not be reproduced, copied, or disclosed without appropriate permission. The commercial use of any information contained in this document may require a license from the proprietor of that information.

THIS DOCUMENT IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES

(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENT, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Table of Contents

1	Introduction	10
1.1	Purpose of this Document	10
1.2	Document Structure	10
2	Alignment of ACTiCLOUD-enabled system libraries with the project's objectives, business scenarios and use cases	12
2.1	Relevance to ACTiCLOUD strategic objectives	12
2.2	Relevance to ACTiCLOUD business scenarios.....	13
	Business scenario 2 - Workload prioritization	13
	Business scenario 3 - Hosting larger workloads.....	13
2.3	Relevance to ACTiCLOUD use-cases.....	13
	Use case 3: Database applications with constant workload and continuous uptime	13
	Use case 4: Database applications with predictable workload burst.....	14
	Use case 6: Analysis of social networks with high dynamism	14
	Use case 7: Enterprise Operations.....	14
3	System Libraries Module Description	15
3.1	Introduction	15
3.2	Guest OS	15
	3.2.1 NUMA Organization	15
3.3	System Libraries special to ACTiCLOUD.....	16
	3.3.1 NC-ALLOC.....	17
3.4	Performance counters.....	20
	3.4.1 Motivation	20
3.5	Other libraries important for ACTiCLOUD systems.....	21
	3.5.1 NC-LAPACK.....	21
	3.5.2 NC-BLACS.....	22
	3.5.3 NC-BTL	23
	3.5.4 NC-MPI	24
4	Improvements to System Libraries	25
4.1	NC-ALLOC improvements in the later phases of the ACTiCLOUD project	25
5	Integration with the ACTiCLOUD stack	26
5.1	NC-ALLOC with MonetDB.....	26
5.2	NC-ALLOC with Neo4j	26
5.3	NC-ALLOC with Hyperscale JVM & JAVA, future work	26

6	Testing and Validation	27
6.1	Testing & Benchmarking	27
6.1.1	Intermediate validation of NC-ALLOC.....	27
6.1.2	NC-ALLOC v3 benchmarks	29
6.1.3	NC-LAPACK: Benchmark Results	30
7	Conclusions, Strategy and Evaluation.....	31
7.1	Conclusion	31
7.2	Next Steps	31
	Appendix A. Documentation / User manuals	32
A.1	NC-ALLOC Setup and Installation procedure	32
7.2.2	A.1.1 NC-ALLOC usage	32
7.3	A.2 NC-LAPACK Usage	33
7.3.1	A.2.1 Compiling an application with NC-LAPACK.....	33
7.3.2	A.2.2 Solving an eigenvalue problem with NC-LAPACK.....	35
7.4	A.3 Numascope and vmxstat usage	37
7.5	A.4 Questions and Answers.....	38
	Appendix B References	39

Figures

Figure 1: Base ACTiCLOUD architecture.	11
Figure 2: NUMA latencies in the memory hierarchy of a large shared memory server from Numascale.	12
Figure 3: Numascale Tools Suite.	17
Figure 4: Pallas benchmark illustrating the performance difference in Open MPI between the Shared Memory BTL and the NumaConnect BTL on a Numascale Shared Memory System.	24

Tables

Table 1: Results for AL benchmark on single server. Configuration: 24 cores (24 HW-threads), 3 x 6828 AMD Opteron, 24 allocator threads. Runtime in seconds.	28
Table 2: Results for AL benchmark on 6-server NUMASCALE system. Configuration: 144 cores / 36 x 6828 AMD Opteron, 72 allocator threads. Runtime in seconds.	28
Table 3: Memory locality (numa-node-local buffer returned) from malloc() in % of calls with the previous configuration.	28
Table 4: Results for AL benchmark on 6-server NUMASCALE system. Configuration: 144 cores / 36 x 6828 AMD Opteron, 144 allocator threads. Runtime in seconds.	29
Table 5: Memory locality from malloc() in % of calls with the previous configuration.	29
Table 6: allocation/deallocation performance test, al.	30
Table 7: allocation/deallocation performance test, pc.	30
Table 8: Eigenvalue solver (N=14000) with NC-LAPACK or MKL.	31

List of Abbreviations

Abbreviation / Acronym	Meaning
CPU	Central processing unit
CSPs	Cloud Service Providers
HPC	High Performance Computing
JVM	Java Virtual Machine
KVM	Kernel-based Virtual Machine
MKL	Intel® Math Kernel Library
NC-ALLOC	NumaConnect™ Memory Allocator
NC-BLACS	NumaConnect™ Basic Linear Algebra Comms System
NC-BTL	NumaConnect™ Byte Transfer Layer
NC-LAPACK	NumaConnect™ Linear Algebra PACKage
NC-MPI	NumaConnect™ MPI emulator
NC-RC	NumaConnect™ Ring Communication
NUMA	Non-Uniform Memory Access
OS	Operating System
ScaLAPACK	Scalable Linear Algebra PACKage
SDN	Software Defined Network
WP	Work Package
ccNuma	cache coherent Non-uniform memory access

1 Introduction

1.1 Purpose of this Document

The Deliverable D3.1 “ACTiCLOUD-enabled system libraries” described the first implementation of the ACTiCLOUD-enabled system libraries. This Deliverable, D3.5 “ACTiCLOUD-enabled system libraries”, reports updates and enhancements to the system libraries done in D3.1

The deliverable is part of WP3, “Evaluation of Databases”. The system libraries are part of the Guest OS in a full stack ACTiCLOUD implementation but can also be used on bare metal in native Linux installations like CentOS 7.6 and Ubuntu 16.04. The use of ACTiCLOUD-enabled system libraries affects the overall performance and stability in a multi-socket NUMA system and performance of application-level databases (MonetDB, Neo4j), language runtime (JVM) and applications using these libraries. Hence, they are important for both strategic objectives of ACTiCLOUD:

- Strategic Objective 1 (SO1): Effective utilization of cloud resources, and more specifically SO1.1: Resource efficiency and SO1.2: Performance stability
- Strategic Objective 2 (SO2): Deployment of resource demanding applications in the cloud (special focus on database applications), especially SO2.1: Scalability in resource provisioning.

The ACTiCLOUD-enabled system libraries resolve inefficiencies and incompatibilities and ensure smooth optimization of memory management, scaling and features in target applications. They optimize memory allocation and OS services within Linux to optimize the relevant facilities of the guest OS. Specifically, the work will include NUMA-sensitive placements of the heap, the stack, and private application memory. Overall, the enabling and optimization mechanisms will also be applicable to virtualization approaches beyond those embraced in ACTiCLOUD (e.g. QEMU/KVM, container-based, or microkernel based). The optimization efforts will be driven by performance analysis of the specific database systems and key applications in ACTiCLOUD to better adapt to their memory requirements and access patterns.

This document is complementary to the software deliverable that includes v2.0 of the ACTiCLOUD-enabled system libraries.

1.2 Document Structure

The document focuses on the modified components in a standard native Linux OS (CentOS 7.6 and Ubuntu 16.04), that enables applications to scale in a large-scale shared memory system¹. We first introduce the System Libraries alignment with ACTiCLOUD’s objectives, business scenarios and use cases. Then, the detailed description of the system libraries themselves is given in Section “System Libraries Module Description”.

As work in ACTiCLOUD is still ongoing we find it natural to follow up with a Strategy and Evaluation section where we discuss open issues, strategy in development and results, before we describe the documentation and further information that can be read to understand the content better.

Figure 1 shows the ACTiCLOUD base architecture. The system libraries are part of the **Guest OS**,

¹ Like those provided from Bull, Numascale, IBM Power and Oracle Scale-up Architecture.

but rely also on lower components, especially the NumaChip support in the bootloader and the kernel. It also relies on a mechanism to communicate the NUMA topology from the kernel to the guest OS. This is handled in the frame of WP2 “Holistic resource management in distributed clouds”.

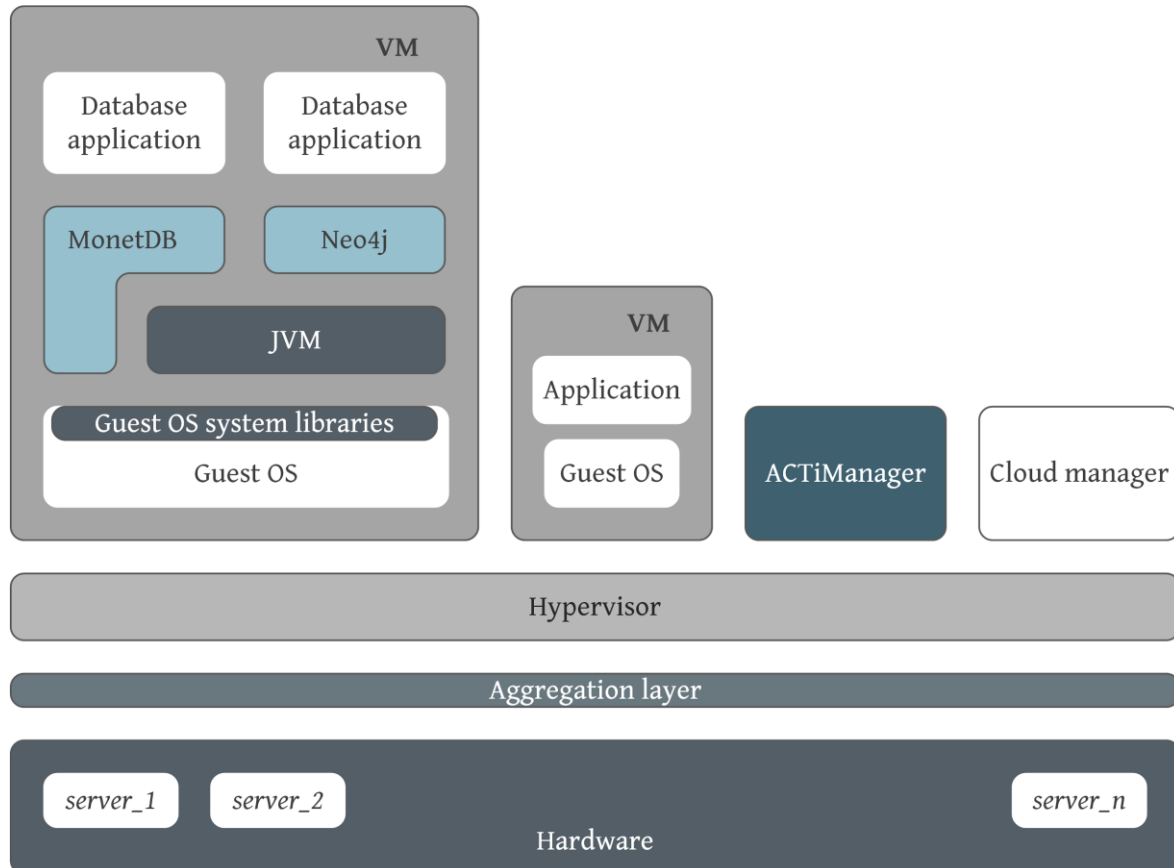


Figure 1: Base ACTiCLOUD architecture.

2 Alignment of ACTiCLOUD-enabled system libraries with the project's objectives, business scenarios and use cases

2.1 Relevance to ACTiCLOUD strategic objectives

In ACTiCLOUD we have two ways of increasing the computing resources that can be made available to single application.

- Scale-Out: Distributing the total load across many computing resources managed by multiple distributed operating systems, and
- Scale-Up: Managing these resources by a single operating system using a Large Shared Memory System.

The major difference from a single CPU server and a Large Shared Memory System is that there are differences in the memory latency. In a ccNuma system, memory access latency differs depending on where the physical memory is located.

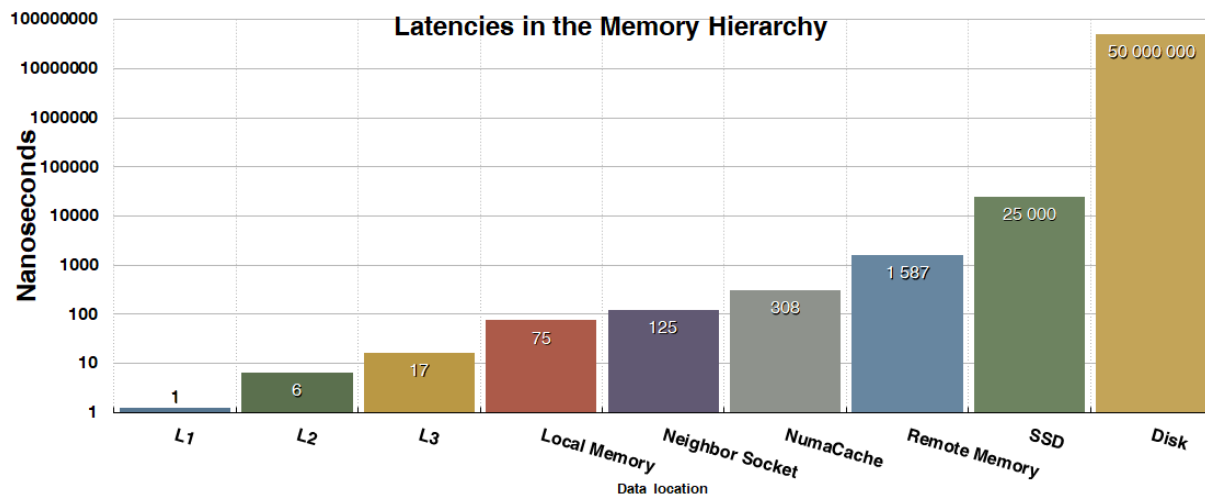


Figure 2: NUMA latencies in the memory hierarchy of a large shared memory server from Numascale.

Although a ccNuma system is logically like a traditional SMP, it is physically different. The runtime of an application is very dependent on where the data is located relative to the running thread. In Figure 2 we compare the different latencies experienced when using a Numascale Shared Memory System. The programming model makes extensive use of caches for faster operations and access to remote RAM faster than access to local disks. What makes this architecture so popular is that if data is not available on the local motherboard but found in the NumaCache it takes 308 nanoseconds to access them (compared to more than a microsecond if we had to access the Remote Memory).

The least efficient memory model for an application on ccNuma system is to have one big array of data on one node that all threads have access to. An optimized application would try to distribute the data as much as possible to be able for threads to use memory local to the core they are running on. In order to be effective, software developers must produce code that is sympathetic to the realities of the new hardware. The System libraries developed in ACTiCLOUD provide a more efficient way to take advantage of the architecture of a large shared memory system using

NUMA. When running large scale applications under heavy load, optimization in system libraries both concerns the strategic objectives related to performance stability and scalability and are important for both strategic objectives of ACTiCLOUD:

- Strategic Objective 1 (SO1): Effective utilization of cloud resources, both SO1.1: Resource efficiency and SO1.2: Performance stability.
- Strategic Objective 2 (SO2): Deployment of resource demanding applications in the cloud (special focus on database applications), especially and specifically SO2.1: Scalability in resource provisioning.

The System libraries developed in ACTiCLOUD provide a more efficient way to take advantage of the architecture of a large shared memory system using NUMA compared to the standard libraries offered in Linux.

2.2 Relevance to ACTiCLOUD business scenarios

Business scenario 2 - Workload prioritization

In a workload prioritization scenario, ACTiCLOUD-enabled system libraries can be utilized to allow high-priority applications that use multiple cores on the Numascale architecture to be allocated to CPUs that are close to each other NUMA-wise. This can be traded-off by moving away “low priority” applications to NUMA nodes that do not necessarily optimize memory access or interfere with the high priority applications.

Business scenario 3 - Hosting larger workloads

Existing platforms do not currently support shared resources for workloads larger than that of a single server in hardware but can only offer scale-out instances that will fit into one of pre-baked instance type. Applications needing large memory spaces, or resource requirements beyond a single machine have been re-engineered to be split into smaller, scalable components. For certain workloads, this works via methods such as map-reduce which can be deployed using current cloud solutions. For other workloads (e.g. large in-memory databases that require online processing of large datasets using more than one server can supply) this approach is not efficient.

There are other solutions out there. The most popular ones are Tidalscale & ScaleMP. These are software-based tools used in clusters that introduces an overhead in latency for remote accesses that spans servers compared to Numascale shared memory servers, as they do not implement a NumaCache or cache coherency. They therefore end up solving remote memory accesses on a page level granularity with a corresponding potential for false sharing and significant increased latency compared with a hardware cc-NUMA solution.

ACTiCLOUD allows CSPs to offer customers dealing with large datasets or large memory demands, VMs capable of expanding beyond a single physical system. The system libraries are sympathetic to the NUMA architecture and key to hosting larger workloads in a scalable way on Scale-up architecture.

2.3 Relevance to ACTiCLOUD use-cases

Use case 3: Database applications with constant workload and continuous uptime

- **Demand:** Performance stability

- **Response:** When running on NUMA architectures we need NUMA-aware system libraries to guarantee continuous performance stability.

Use case 4: Database applications with predictable workload burst

- **Demand:** Scalability and elasticity in resource provisioning
- **Response:** A large Numascale shared memory system can handle large bursts of work well if the system libraries distribute the workload in an intelligent fashion. This type of system provides great scale-up scalability.

Use case 6: Analysis of social networks with high dynamism

- **Demand:** Scalability and elasticity in resource provisioning
- **Response:** Large shared memory systems enable graph databases like Neo4J to fit much larger graphs in-memory. This requires scalable system libraries for both Neo4j and the underlying JVM (including efficient garbage collection).

Use case 7: Enterprise Operations

- **Demand:** Large enterprise applications expect full value from the hardware when scaling up.
- **Response:** The NUMA-aware system libraries support this need.

3 System Libraries Module Description

3.1 Introduction

This section describes the optimizations and modifications to system libraries in the guest OS in ACTiCLOUD. These are mainly related to large shared memory systems with different access times depending on where a CPU and the memory it accesses is placed. We call them Numa-Aware optimizations. We start by describing the Guest OS' considerations in order to create a context for the system libraries we have worked on in ACTiCLOUD and the rationale for creating a framework for monitoring performance across servers highlighting Numa-Aware placement of threads, applications and VMs.

3.2 Guest OS

This section describes the integration of a typical Guest OS used in ACTiCLOUD, i.e., the virtualized environment that is provided by ACTiCLOUD. The system libraries and the JVM run within the Guest OS.

The Guest OS is not modified within the ACTiCLOUD project. In this way ACTiCLOUD-enabled systems will be able to operate with any type of Guest OS. However, our experimentation and any changes to the system libraries of the Guest OS are performed on Linux-based systems. Hence, we focus our effort on Linux systems, and particularly on CentOS 7 and Ubuntu 16.04 for the prototype work.

The Guest OS includes important system libraries. For instance, *glibc*² is the GNU Project's implementation of the C standard library and provides macros, type definitions and functions for tasks such as memory management, string handling, mathematical computations, input/output processing, and several other operating system services. Another example is *libNUMA*³ that offers a simple programming interface to the NUMA policies supported by the Linux kernel. Available policies are page interleaving, preferred node allocation, local allocation, or allocation only on specific nodes. With *libNUMA* it is also possible to bind tasks to specific nodes statically. Finally, one more example is Open MPI that is an open source implementation of the Message Passing Interface. MPI is traditionally used with processes on scale-out servers but using ACTiCLOUD libraries NC-BTL and NC-MPI we can avoid the time-consuming overhead of protocol management needed by MPI, as we convert the MPI messages to threads to enable new level of caches in NC-BTL. In NC-MPI we convert the MPI messages to Open MP messages and avoid the overhead in the MPI protocol.

In ACTiCLOUD, the default system libraries, that come either pre-installed with the base image or are directly installed from the official distribution repositories, are extended to provide enhanced capabilities in memory allocation, memory management, and topology awareness within the Guest OS.

3.2.1 NUMA Organization

The command `numactl --hardware` gives information about the organization of the NUMA nodes and the memory latencies. The output provides a map of the inventory of available NUMA

² <https://www.gnu.org/software/libc/>

³ <https://github.com/numactl/numactl/blob/master/libnuma.c>

nodes in the system and different relative latencies between specific NUMA nodes (node distances, lower is better).

```
[nscale@node1 ~]$ numactl --hardware
available: 36 nodes (0-35)
node 0 cpus: 0 1 2 3
node 0 size: 61402 MB
node 0 free: 57547 MB
node 1 cpus: 4 5 6 7
node 1 size: 64511 MB
node 1 free: 53773 MB
...
node 35 cpus: 140 141 142 143
node 35 size: 64500 MB
node 35 free: 62816 MB
node distances:
node  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
0: 10 16 16 22 16 22 100 100 100 100 100 100 100 100 100 100 100 100
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100
100 100 100 100
1: 16 10 22 16 22 16 100 100 100 100 100 100 100 100 100 100 100 100
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100
100 100 100 100
2: 16 22 10 16 16 22 100 100 100 100 100 100 100 100 100 100 100 100
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100
100 100 100 100
3: 22 16 16 10 22 16 100 100 100 100 100 100 100 100 100 100 100 100
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100
100 100 100 100
4: 16 22 16 22 10 16 100 100 100 100 100 100 100 100 100 100 100 100
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100
100 100 100 100
5: 22 16 22 16 16 10 100 100 100 100 100 100 100 100 100 100 100 100
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100
100 100 100 100
6: 100 100 100 100 100 100 100 10 16 16 22 16 22 100 100 100 100 100
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100
100 100 100 100
...
35: 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 22 16
22 16 16 10
```

3.3 System Libraries special to ACTiCLOUD

This component provides enhanced capabilities in memory allocation and topology awareness within the guest operating system.

The stack shown in Figure 3 summarizes the tools and dependencies available to run an application optimally on a scale-up server. The libraries, NC-LAPACK, NC-BTL and NC-MPI are used for optimization of workloads running on the ACTiCLOUD platform, but also validation

benchmarks showing scaling on a NUMA system and between different NUMA systems. Benchmarking of these libraries can be compared with other Numascale systems to show that they are performing as expected and will be used to increase performance for HPC and Big Data applications. Although NC-LAPACK, NC-MPI, NC-BLACS, NC-RC and NC-BTL help exploit NUMA awareness in large SMPs and are used for validation benchmarks and performance improvement in scale-up architectures, the main system library we develop and use in ACTiCLOUD is NC-ALLOC.

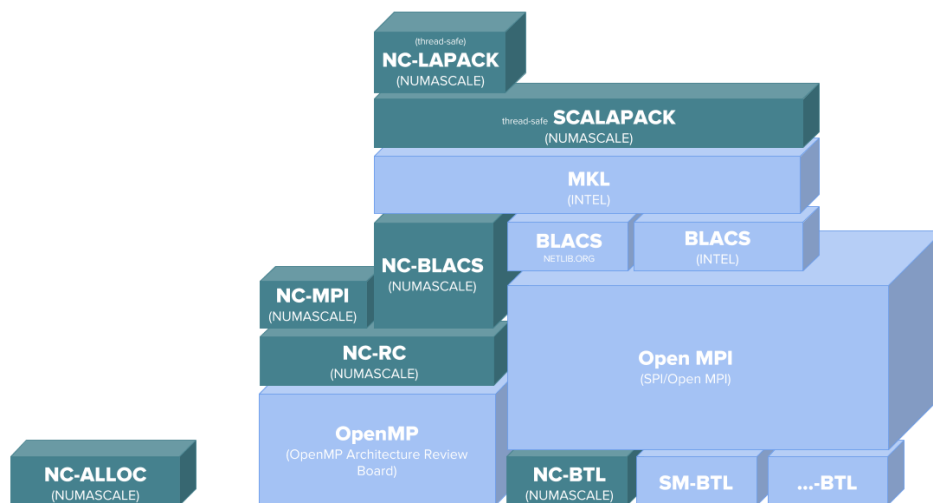


Figure 3: Numascale Tools Suite.

3.3.1 NC-ALLOC

NC-ALLOC is a memory allocator alternative to using the libc built-in. It hooks to the malloc(), free(), realloc() and memalign() functions in libc. There are several allocators which can be used instead of the standard built-in allocator. Some of them are for general use and others are optimized for special demands of the application. A well-known allocator is Hoard⁴. An allocator for general use must address speed, scalability, fragmentation and avoidance of false sharing and lock contention. Hoard does this very well, but it does not assign every thread to its own private heap. It also reuses empty memory blocks in other threads. This may result in returning remote memory from a malloc() call, which is suboptimal in a NUMA architecture.

NC-ALLOC may be up to 10x faster than the standard memory allocator. It is also faster on large systems than for example the Hoard memory allocator. Consequently, an application will benefit from the same speedup, if it uses many threads and has a lot of memory management operations.

⁴ <http://hoard.org/>

3.3.1.1 NC-ALLOC Design Challenges

The motivation behind creating NC-ALLOC is the ability to scale-up better than other memory managers. When many threads in a NUMA system operate on a shared memory, then we need special consideration regarding locality and latency, avoiding contention. On the other hand, we do not need to be as restrictive on memory usage as traditional memory managers.

The main challenges we have addressed in ACTiCLOUD are therefore:

- Allow runtime tuning
- Avoid time consuming kernel calls, like calls to the OS that require user space to kernel space switching (and consequent cache flushes)
- Efficiently avoid false sharing as this generates coherency traffic between NUMA nodes (worst case between servers)
- Minimize inter-thread locks, that are slow operations
- Reduce time searching for free memory
- Have as little fragmentation as possible to maintain efficient memory utilization

One major difference between other allocators and NC-ALLOC is that in order to avoid additional latency in inter-thread locks, and coherency traffic we have decided that NC-ALLOC will manage one heap per thread.

Avoid time consuming kernel calls

One of the challenges of every allocator is to allocate as little memory from the OS and to call the OS as rarely as possible. So, the allocator tries to pack smaller blocks in the allocated chunks. In a special allocator with only a malloc() but no free() call, this can be done without fragmentation. But in a general-purpose allocator fragmentation cannot be avoided. This results in less efficient use of memory.

Fragmentation compromise and bins

When introducing one heap per thread (instead of letting threads share heaps as the standard memory allocator of libc does) this affects fragmentation drastically. NC-ALLOC uses two different routines for allocating small and big memory blocks. Blocks bigger than or equal to 2 MB (huge page size) are considered big blocks, while the rest are considered as small blocks. Big blocks get allocated directly from the OS while small blocks are managed inside 2 MB chunks. NC-ALLOC manages a list of bins of different size. Instead of using power of two sizes (as the standard memory allocator does) it increases the size of bins by power of two half: 16, 24, 32, 48, 64, 96, 128, 192, 256, ..., 1MB. This results in less fragmentation when compared to the standard memory allocator approach, and we need that as we use many more heaps. For example, if 150 bytes get allocated inside a 192-byte bin, 42 bytes are wasted. If they are allocated inside a 256-byte bin, 106 bytes would be wasted.

The use of bins instead of tree structures avoids the need for searching free blocks in the tree but may have more overhead in the bin gaps because adjacent blocks don't get combined to bigger blocks. The challenge is to determine very quickly the correct bin size from the allocation size. The granularity of the returned blocks in a malloc(), memalign() or realloc() is 16 bytes. So, it is safe to use up to e.g. 64 bytes even if the requested size was only 49 bytes. Additionally, it can be more efficient using a 4 byte or 8-byte access when accessing byte 49.

False Sharing

NC-ALLOC manages one heap per thread. A heap only gets created when a thread for the first time calls `malloc()` or `memalign()`. Data structures used by `free()` and `realloc()` may be accessed by all threads. This results in shared cache lines. The allocated blocks are used exclusively by the owning thread. This avoids false sharing, which leads to lock operations when more threads share a heap, like with `libc` and `Hoard`. False sharing occurs when two or more threads share the same cache line. This could result in coherency traffic which is very expensive when running large NUMA systems. When a chunk gets returned to the OS, it might get allocated by another thread. This is determined by the memory allocation policy of the OS. When a block gets returned in a `free()` call by a different thread (inter-thread free call), this block needs to be returned to the owning thread inside NC-ALLOC.

Avoid expensive inter-thread locking

One more challenge to all allocators is to run with as little locking as possible to avoid lock contention. Since `free()` calls can be done from a different thread, all `malloc()`, `memalign()`, `realloc()` and `free()` calls need to use a lock. NC-ALLOC solves this by avoiding inter-thread calls, due to the use of one heap per thread. If inter-thread free calls are avoided this lock costs are nearly in zero time. After an inter-thread free call, the cache line containing the lock needs to be synchronized by the two using threads. In a producer-consumer scenario where one thread creates (allocates) messages and sends them to other threads, this locking can get time consuming but is unavoidable.

Avoid spending time searching through long lists

An allocator should process `malloc` and `free` calls in nearly constant time, so the time should not increase when more and more blocks are in use or fragmentation is high. It should not search through lists. NC-ALLOC addresses this issue by never searching lists. When a big block gets freed and is not returned to the OS immediately, it becomes the head of a list and is reused by the next allocation call from a thread running on the same NUMA node. These lists exist per NUMA node. All free blocks are reused in a LIFO (last in first out) way to increase memory locality and cache usage.

Freeing large blocks and runtime tuning

NC-ALLOC allocates blocks of 2MB or larger from the OS by `mmap()` calls. It aligns the blocks on a 2 MB boundary by extending the block size by up to 2 MB, adjusting the starting address inside the allocated block and returning the head and tail fragments to the OS. This way it makes sure that transparent huge pages are used. NC-ALLOC returns blocks to the OS if the number of free pages exceeds a configuration threshold. The number of unused big blocks to keep mapped is 4 and the total number of unused blocks to keep mapped is 16. These are compile-time constants. NC-ALLOC hooks `pthread_exit()`, frees the threads heap and returns all blocks from this heap to the OS. In case a thread ends without calling `pthread_exit()`, its heap only gets freed when the process ends.

The statistics and debug version of NC-ALLOC (`libncallocs.so`) uses per thread and global heap usage counters and dumps this information to `stdout` at process end. This solves the technical challenge in gathering this information at runtime with nearly no runtime cost.

3.4 Performance counters

3.4.1 Motivation

When developing cache-coherent interconnects, there are no universal mechanisms available to capture interconnect resource utilization. Likewise, when developing and tuning software applications on systems large enough to warrant cache-coherent interconnects, existing mechanisms to collect resource usage are limited to the system's core and single core performance counters.

High-rate capture of fine-grained on-chip interconnect counters would present useful telemetry both to guide interconnect and application development and tuning. As an example of the first case, profiling expensive on-chip buffering would allow allocating just enough buffer space to deliver maximum throughput (based on the round-trip and number of outstanding cycles) and to understand if and what parts of the interconnect behavior bottleneck the system. In the latter case, the impact of algorithmic adjustments can be understood at a cache-coherent and high-level NUMA viewpoint.

To wit, ICCS has published a paper regarding “Performance Prediction of NUMA Placement: A Machine-Learning Approach”⁵, that presents models for predicting the impact of memory placement on application performance.

A performance counter framework is especially interesting for users of large shared memory systems like the Numascale system in ACTiCLOUD, where there are more complex NUMA topologies and bigger penalties when data placement and processing crosses higher latency and cost paths.

Since Numascale designs processor interconnect silicon that enables somewhat larger topologies by combining multi-socket servers, they are interested in supporting the users with counters specific for the cache coherent connections between the physical servers. Numascale has delivered the *vmxstat* tool that performs convenient event monitoring on the command line, in the style of the *vmstat* command. The *numascope* tool has been developed for detailed visualisation.

UNIMAN is using Numascope in order to profile JVM workloads, UMEA is evaluating it from the perspective of decision making including where to place VMs, while ICCS is using it in understanding better and predicting more accurately the impact of NUMA in application scenarios.

Both tools allow the user to specify the resolution of data capture, and support annotation of the traces by writing into a user-writable FIFO (named pipe).

Numascope⁶ provides real time HTML5 graphing. To illustrate the graphical interface, numascope was executed while the NASA Parallel Benchmarks (NPB) Lower Upper (LU) matrix transposition benchmark was running:

⁵ Fanourios Arapidis, Vasileios Karakostas, Nikela Papadopoulou, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. "Performance Prediction of NUMA Placement: A Machine-Learning Approach". In 1st International Workshop on Next Generation Clouds for Extreme Data (XtremeCLOUD 2018) - held in conjunction with CloudCom 2018. <https://acticloud.eu/dissemination/publications>

⁶ Appendix A.3 describes the usage of *vmxstat* and *Numascope*.



Here, with AMD HyperTransport Read Block commands being counted per-server, we can observe a highly imbalanced workload that stresses the interconnect from different servers at different times. The dynamic range is around a factor of 1000 (4500 to 4.5M events per second). The figure shows that the counted events from the different physical servers (in the 6-server single image partition) are showing some overlapping in utilizing their NumaConnect links (each different color represent one physical server). In order to mitigate imbalance, the software designer can choose several approaches e.g.:

- Redesign the application to do more balanced communication.
- Use more local access.
- Utilize caches.
- Consider a well-suited programming paradigm that favors thread parallelization like Open MP, or pthreads.
- Use threads instead of processes in order to utilize the per server NumaCache.
- If it is a linear algebra application switch to ScaLAPACK⁷ and use with NC-BLACS.
- If it is an MPI application switch to a threaded version of the MPI implementation that allows for higher cache utilization, e.g. NC-MPI.

Kernel event counters are included to allow observing Virtual Memory activity that would stall application progress.

3.5 Other libraries important for ACTiCLOUD systems

3.5.1 NC-LAPACK

The most relevant library apart NC-ALLOC is the popular NC-LAPACK. LAPACK⁸ can solve systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems. LAPACK can also handle many associated computations such as matrix factorizations

⁷ <https://www.netlib.org/scalapack/>

⁸ <https://www.netlib.org/lapack>

or estimation of condition numbers. This is relevant to popular data analytics applications using the databases in ACTiCLOUD that can store their information in memory instead of inefficient disks, e.g. in Monte Carlo simulations.

LAPACK contains:

- **driver routines** for solving standard types of problems
- **computational routines** to perform a distinct computational task
- **auxiliary routines** to perform a certain subtask or common low-level computation

Each driver routine typically calls a sequence of computational routines. Taken as a whole, the computational routines can perform a wider range of tasks than are covered by the driver routines.

3.5.1.1 Making LAPACK NUMA-AWARE

Building a threaded version of ScaLAPACK

Like LAPACK, ScaLAPACK is a library of linear algebra routines: it solves dense and banded linear systems, least squares problems, eigenvalue problems, and singular value problems. But compared to LAPACK, ScaLAPACK is **designed for parallel distributed memory machines**.

Based **only** on the complex **ScaLAPACK** library for running on parallel distributed memory machines, the **NumaConnect Linear Algebra PACKage** (NC-LAPACK) provides subroutines whose simple calls reminds those of LAPACK, if not the same, making NC-LAPACK a much simpler library to use than ScaLAPACK.

Moreover, for NC-LAPACK we made some minor modifications to make the standard ScaLAPACK thread-safe.

This was done by declaring all global variables in TLS (thread local storage) with the `__thread` attribute. There were also some static local variables which were renamed to make them globally unique, moved to global scope and declared in TLS.

In ACTiCLOUD, NC-LAPACK will be used to run linear algebra applications and verification of the performance of the Numascale platform, natively, and on top of the Guest OS in different sized Virtual Machines.

This implementation makes LAPACK scalable. To optimize for a NUMA system, we replaced the Message Passing Interface protocol overhead that is not needed on a Numascale Shared Memory System, or similar large NUMA architectures. This was the design reason behind NC-BLACS.

3.5.2 NC-BLACS

NC-BLACS allows applications to scale linear algebra applications on the multicore ACTiCLOUD partitions (SO2.1: Scalability in resource provisioning).

The vendor optimized LAPACK routines in MKL or ACML libraries support parallelism through OpenMP. However, these routines showed low scaling in a distributed memory system. ScaLAPACK instead uses blocked algorithms. It decouples the problem as well as possible and solves smaller problems in parallel.

ScaLAPACK uses the BLACS (Basic Linear Algebra Communication System) library. In ScaLAPACK there are no direct MPI calls or other ways of inter-process communication. The BLACS library allows to send or broadcast parts of matrices between the tasks, using MPI as the underlying

communication library. BLACS also provides synchronization through barriers and provides some simple reduction operations.

Building a version of BLACS using threads instead of MPI processes

NC-BLACS is an OpenMP implementation of BLACS, where we have implemented a way to allow most MPI applications (which are usually distributed over multiple processes) through a BLACS wrapper, to run distributed over multiple threads. NC-BLACS therefore runs an MPI application in a single process over multiple threads using OpenMP. This removes the communication overhead of using processes and improves NUMA awareness. We have put the core part that emulates MPI communication over OpenMP in a separate library called NUMA-Connect Ring Communication (NC-RC). This library is the main dependency of NC-MPI and NC-BLACS.

3.5.3 NC-BTL

NC-BTL provides better scalability for HPC applications running on large VMs in ACTiCLOUD (SO2.1: Scalability in resource provisioning).

NC-BTL stands for NumaConnect BTL (Byte Transfer Layer). NC-BTL is used to improve any Open MPI application that runs with the Numascale fabric: **NumaConnect**. As such we have used the BTL layer in OpenMP to create a module that respects the NUMA topology of our system.

NC-BTL is only compatible with Open MPI; it will not work with MPICH nor with other MPI flavors. It provides a NumaConnect specific BTL module for Open MPI to exploit features of the NumaConnect chip to gain the best possible bandwidth and low latency. NC-BTL accelerates significantly Open MPI, for some workloads and sizes achieves a speedup of 23, as shown in Figure 4. In ACTiCLOUD, NC-BTL will be used to run MPI applications for verification and users that wants to use MPI.

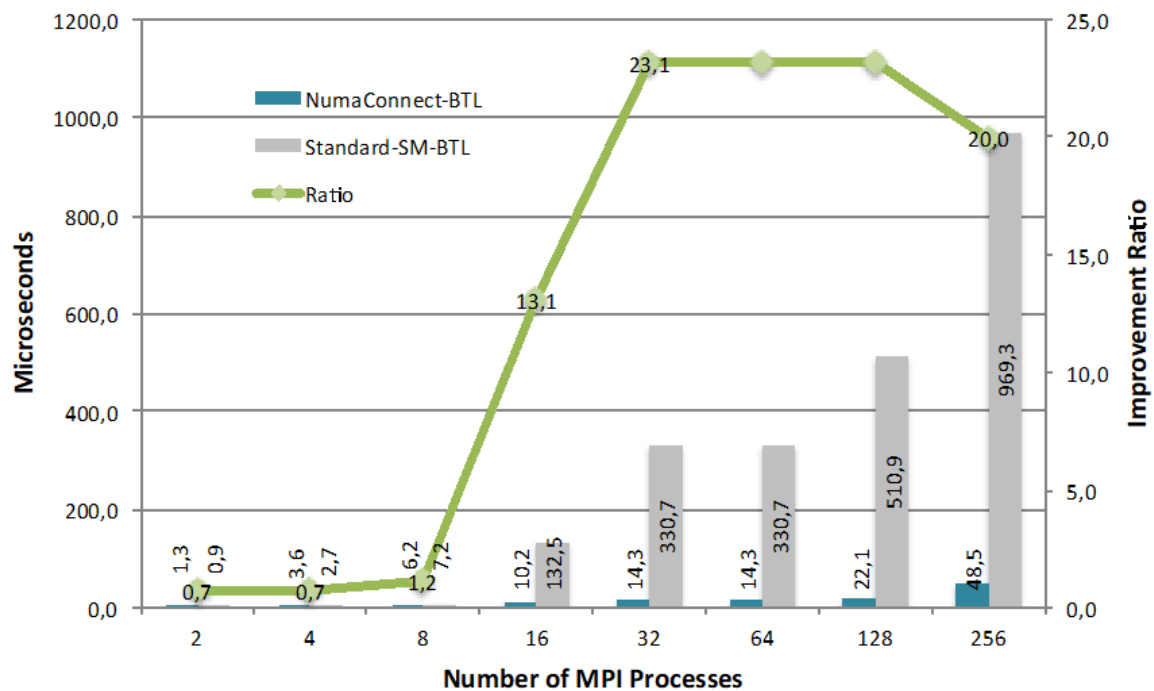
MPI_Barrier, 2 - 256 processes

Figure 4: Pallas benchmark illustrating the performance difference in Open MPI between the Shared Memory BTL and the NumaConnect BTL on a Numascale Shared Memory System.

3.5.4 NC-MPI

NC-MPI provides better scalability for HPC applications running on large VMs in ACTiCLOUD (SO2.1: Scalability in resource provisioning).

NC-MPI is an MPI emulation doing inter-threads communication instead of inter-process communication. NC-MPI is a subset of MPI 1. It includes about 60 of the most common MPI routines which instead of using MPI for communication will run with multiple threads within a single process with OpenMP. The core part that emulates MPI communication over OpenMP is packaged in a separate library called NUMA-Connect Ring Communicator (NCRC) library.

4 Improvements to System Libraries

4.1 NC-ALLOC improvements in the later phases of the ACTiCLOUD project

During evaluation with ACTiCLOUD partners we found some limitations in NC-ALLOC. The database applications created and terminated many threads. These threads did not release all memory before termination. This is acceptable behavior, because the memory might be still used by other threads. But NC-ALLOC was designed for HPC applications where threads normally get created only once (but often) and terminate at process end. NC-ALLOC did not release the thread's heap until all memory was freed. This often resulted in an accumulation of heaps and a termination when the limit was reached. Hoard and TCMalloc (part of gperftools, the google performance tools, <http://goog-perftools.sourceforge.net/doc/tcmalloc.html>) free memory to a pool which exists independent of the threads private heaps. NC-ALLOC now also uses additional heaps, each associated with its NUMA node instead of one global pool or heap. This keeps the memory block local and reduces false sharing between NUMA nodes. These blocks may be reused by another heap on the same NUMA node. So false sharing might occur with NC-ALLOC but is limited to cores of the same NUMA node, since this has a very small performance penalty.

Every allocation requires locking of the heap of the calling thread. Every deallocation requires locking of the heap of the owner of the memory. Deallocation by a thread not owning the memory can be very costly because the 'foreign' thread needs to modify the heap structures of another thread. So far NC-ALLOC used one lock per thread. Now every bin (list of fixed size chunks) is individually locked. This more fine-grained locking allows parallel access to different bins and reduces false sharing in case of foreign access. Foreign access is common in producer/consumer scenarios where one thread allocates a message and sends it to another thread which then deallocates it.

Originally, NC-ALLOC was able to gather statistics depending on the value of an environment variable. Even if statistics gathering was not enabled it still had a little performance cost. To avoid paying this overhead there are now two versions of NC-ALLOC, `libncalloc.so` and `libncallocs.so`. The first is the minimal version which is compiled for performance and does not have the ability to gather statistics. The second is the statistics version. It always gathers statistics information and prints it to stdout after process termination.

The major changes are:

- per NUMA node heap independent of threads private heap
- support for an unlimited number of threads (avoiding max threads assertion seen with neo4j/MonetDB)
- Support for unlimited thread creation/termination
- Release of memory after owning thread termination
- Fine grained heap locking
- Statistics/Debug version and minimal version of `libncalloc[s].so`
- Deployed to Athens and Oslo system

5 Integration with the ACTiCLOUD stack

5.1 NC-ALLOC with MonetDB

Numascale and MonetDB did a thorough testing of system libraries in “D4.3: ACTiCLOUD Intermediate Evaluation”. Since then, Numascale have started profiling MonetDB with and without NC-ALLOC using the SF1000 dataset, which comprises about 1TB of data, in order to see if thread/process placement and pinning will improve the results.

5.2 NC-ALLOC with Neo4j

NEO4J did a thorough testing of system libraries presented in “D4.3: ACTiCLOUD Intermediate Evaluation”. Numascale, UNIMAN and NEO are profiling and optimizing the performance of application layer (Neo4J, MonetDBLite-Java), middleware/JVM Layer (HyperScaleJVM) and hardware layer (NUMASCALE) in order to scale-up neo4j on a Numascale shared memory system. Our experiments so far have shown that NC-ALLOC cannot benefit JVM-based applications, see 5.3 for more details.

5.3 NC-ALLOC with Hyperscale JVM & JAVA, future work

UNIMAN is working with Numascale and NEO in scaling the performance of HyperScale JVM on a Numascale system. Evaluation of Java workloads, namely the DaCapo benchmark suite⁹, has shown that the use of NC-ALLOC does not impact the performance of JVM-based workloads. This is attributed to the lack of malloc/free invocations in the JVM. The JVM allocates big chunks of memory, using mmap instead of malloc, and manages them with its own algorithms that are tightly coupled with the garbage collection (GC) algorithms in use. In ACTiCLOUD we are exploring ways to make the memory management algorithms of JVMs NUMA-aware. One potential approach that we are looking into is the integration of NC-ALLOC logic in the JVM memory management and garbage collection algorithms.

⁹ <http://dacapobench.sourceforge.net/>

6 Testing and Validation

6.1 Testing & Benchmarking

6.1.1 Intermediate validation of NC-ALLOC

A thorough validation of the NC-ALLOC compared to tcmalloc and libc was delivered in “D4.3: ACTiCLOUD Intermediate Evaluation”, Section 3.3. It is repeated here for public sharing:

AL Benchmark

The AL (AL is short for allocator) benchmark is a synthetic benchmark created by Numascale that mimics the operations from real applications in an aggressive way to highlight memory latency.

AL runs several threads. They allocate 10000 chunks of random size between 1 byte and 4 MB. All allocated chunks then are written using memset to force the memory to commit. Then, AL frees half of the chunks and allocates them again. Finally, all chunks are freed. This sequence is repeated 10 times. The memory access is restricted to the allocating thread, so no thread is accessing chunks of a different thread. This is the ideal access pattern of a NUMA aware application. The lines marked with "no memset" show results where the allocated chunks were not committed by memset. This allows comparing the speed of the allocation itself. A few bytes used as a header prepending every allocation are still committed though.

Distribution of allocations based on allocation size:

size count

[0..1]	188060	[2K..4K]	155710
[1..2]	187730	[4K..8K]	147170
[2..4]	377000	[8K..16K]	68400
[4..8]	754870	[16K..32K]	1960
[8..16]	1530690	[32K..64K]	3500
[16..32]	3060120	[64K..128K]	8896
[32..64]	6098210	[128K..256K]	16650
[64..128]	6908510	[256K..512K]	20770
[128..256]	248310	[512K..1M]	9290
[256..512]	502530	[1M..2M]	2160
[512..1K]	942760	[2M..4M]	4080
[1K..2K]	212910		

6.1.1.1 Evaluation of NC-ALLOC on the ACTiCLOUD system with the AL Benchmark**Single server**

Table 1: Results for AL benchmark on single server. Configuration: 24 cores (24 HW-threads), 3 x 6828 AMD Opteron, 24 allocator threads. Runtime in seconds.

Size	Libc	TCMalloc	NC-ALLOC
[1B..100KB]	5.4	4.5	4.2
[1B..1MB]	14.5	4.7	4.3
[1B..4MB]	10.4	4.8	4.4
[1B..4MB] (no memset)	7.0	4.6	4.2

6-Server NUMA-Connect system – 72 threads

Table 2: Results for AL benchmark on 6-server NUMASCALE system. Configuration: 144 cores / 36 x 6828 AMD Opteron, 72 allocator threads. Runtime in seconds.

Size	Libc	TCMalloc	NC-ALLOC
[1B..100KB]	8.0	12.8	5.0
[1B..4MB]	21.5	20.1	5.2
[1B..4MB]	23.2	22.7	5.5
[1B..4MB] (no memset)	9.6	11.4	5.1

Table 3: Memory locality (numa-node-local buffer returned) from malloc() in % of calls with the previous configuration.

	Libc	TCMalloc	NC-ALLOC
local NUMA node	100.0	4.3	99.2
local server	100.0	18	100.0

6-Server NUMA-Connect system -144 threads**Table 4:** Results for AL benchmark on 6-server NUMASCALE system. Configuration: 144 cores / 36 x 6828 AMD Opteron, 144 allocator threads. Runtime in seconds.

Size	Libc	TCMalloc	NC-ALLOC	NC-ALLOC with NUMA pool
1B-100KB	12.6	30.0	6.0	6.0
1B-4MB	43.4	53.3	6.9	7.0
1B-4MB (no memset)	15.8	23.6	6.4	6.5

Table 5: Memory locality from malloc() in % of calls with the previous configuration.

	Libc	TCMalloc	NC-ALLOC	NC-ALLOC with NUMA pool
local NUMA node	100.0	4.3	99.7	99.7
local server	100.0	18	100.0	100.0

The benchmark results show that libc does not perform as well as NC-ALLOC even though it returns 100% local memory. It can be assumed that libc is NUMA aware and migrates pages between the nodes when it detects an allocation call from a remote node. But the performance is still low because cache lines in a shared state if reused by a different thread. This results in heavy probing traffic in this cc-NUMA system. Since the thread which used the page before freed the memory it is in the shared state and the probing traffic is unnecessary.

6.1.2 NC-ALLOC v3 benchmarks

In order to compare the performance of NC-ALLOC with the standard memory allocator, Numascale have created two synthetic benchmarks: AL and PC. AL and PC are small programs that can be found under `/opt/ncalloc` on the Large Shared Memory Systems in ACTiCLOUD:

- **al:** this program uses OpenMP to run on many threads and allocates/deallocates chunks of memory. The allocated memory is committed and returned by owning thread only. Results are shown in Table 1.
- **pc:** this program uses OpenMP like al, but all threads with even thread id act as message producers (allocators) and all threads with uneven thread id act as message consumers (deallocators). So the consumers need to return memory to a 'foreign' threads heap. Results are shown in Table 2.

Table 6: allocation/deallocation performance test, al.

# cores	core IDs	Test duration with NC-ALLOC (seconds)	Test duration w/ glibc allocator (seconds)	NC-ALLOC speed-up factor
144	0-143:1	4.9	42.0	8.5
72	0-143:2	2.4	18.5	7.7
72	0-71:1	2.8	22.3	7.9
48	0-143:3	5.3	12.6	2.3
48	0-47:1	2.2	20.6	9.3
36	0-143:4	1.8	8.8	4.8
36	0-71:2	1.7	11.6	6.8
36	0-35:1	2.0	15.2	7.6

The test, Table 1 shows speed-ups, from more than 2x up to 9x, that are also proportional to the number of threads.

Table 7: allocation/deallocation performance test, pc.

# cores	core IDs	Test duration with NC-ALLOC (seconds)	Test duration w/ glibc. allocator (seconds)	NC-ALLOC speed-up factor
144	0-143:1	130	137	1.05
72	0-143:2	36	32	0.89
72	0-71:1	26	27	1.04
48	0-143:3	11.1	10.4	0.94
48	0-47:1	8.6	8.0	0.93
36	0-143:4	6.4	6.7	1.05
36	0-71:2	6.2	6.0	0.97
36	0-35:1	4.7	4.6	0.98

To summarize, NC-ALLOC often improves the performance of an application by a varying factor, but very rarely introduces any performance loss.

6.1.3 NC-LAPACK: Benchmark Results

Table 3 is a summary of a few more setups, comparing NC-LAPACK against MKL (Intel Math Kernel Library, the Intel optimized version of LAPACK) on the same calculation. NC-LAPACK demonstrates very good speed-up factors increasing with the number of cores used.

Table 8: Eigenvalue solver (N=14000) with NC-LAPACK or MKL.

Total #cores used	Repartition of the cores	NC-LAPACK Runtime (minutes)	MKL Runtime (minutes)	NC-LAPACK speed-up factor
144	[0-143:1]	7.9	363.2 (6h)	45.8
48	[0-47:1]	10.3	267 (4.4h)	25.9

7 Conclusions, Strategy and Evaluation

7.1 Conclusion

This deliverable contains information about the implementation of NC-ALLOC and other important NUMA aware libraries like NC-LAPACK and how we have used test applications to show scaling in Chapter 4.

Given that the ACTiCLOUD partners' workloads benefit a lot from Numascope performance tracing there is a large common effort in this direction in the last part of the project.

7.2 Next Steps

Numascale, UNIMAN, NEO and ICCS will continue to optimize workloads based on NC-ALLOC and Numascale performance counters. Tuning, design changes and implementation improvements are expected to reach this goal.

Appendix A. Documentation / User manuals

A.1 NC-ALLOC Setup and Installation procedure

NC-ALLOC is presented as a precompiled binary together with two benchmark validation applications. In ACTiCLOUD, we plan to add extensive runtime support to optimize applications and databases with special needs.

```
[root@node1 ncalloc]# ls
libncalloc.so  libncallocs.so  al  pc
[root@node1 ncalloc]# pwd
/opt/ncalloc
[root@node1 ncalloc]#
```

7.2.2 A.1.1 NC-ALLOC usage

In order to use NC-ALLOC with an application, the user can either link to the library or preload it as shown below:

```
export LD_PRELOAD=/opt/ncalloc/libncalloc[s].so
```

The debug/statistics version of NC-ALLOC prints out statistics when the library gets unloaded, e.g., at process end. You can also configure how many free pages get cached by each heap before they are returned to the OS, through the environment variables `NCALLOC_CACHE_PAGES`, `NCALLOC_CACHE_BIGBLOCKS` and `NCALLOC_CACHE_MEM`.

```
# the default is 64 pages
export NCALLOC_CACHE_PAGES=<num>
# the default is 64 blocks
export NCALLOC_CACHE_BIGBLOCKS=<num>
# the default is 1G (1 GByte) per heap
export NCALLOC_CACHE_MEM=<num><m|M|g|G>
```

The memory binding policy can also be configured by:

```
export NCALLOC_MPOL=[MPOL_PREFERRED | MPOL_BIND]
```

the default being `MPOL_PREFERRED`.

then simply see how opening and closing `vi` would tell on its own memory allocation:

```
heap    0, mapped    28 MB, allocated    0 MB, malloc time 2105 us, free time    32
us

total combined values for all threads :

malloc calls    741
free calls      59
realloc calls    0
memalign calls  0
create heap calls    1
exit heap calls      0
inter thread free calls 0
mmap() calls      14
```



```

unmap() calls          0
malloc time            2105 us
free time              32 us

<=1 <=2 <=4 <=8 <=16 <=32 <=64 <=128 <=256 <=512 <=1K <=2K
  1   22   15   81  176  205  155   26   13   31   10   6

```

7.3 A.2 NC-LAPACK Usage

7.3.1 A.2.1 Compiling an application with NC-LAPACK

Remember the aim of NC-LAPACK is to mimic the simplicity of LAPACK's routines while calling ScaLAPACK's routines instead. Therefore, as for ScaLAPACK, NC-LAPACK does not make direct calls to neither OpenMP nor MPI, all are based on BLACS (or NC-BLACS for the matter).

- Running NC-LAPACK on top of BLACS is not supported yet, but it would simply run a simpler implementation of ScaLAPACK on top of MPI/OpenMP.
- Running NC-LAPACK on top of NC-BLACS will run the same genuine implementation of ScaLAPACK, but entirely on top of OpenMP, using NC-RC and NC-BLACS to handle the MPI calls.

Assuming an application contains LAPACK routines and the user wants to compile and run it with NC-LAPACK instead of MKL, the NC-LAPACK library needs to be linked.

An example lies under `/opt/ncblacs/EIG_TEST`, where we use the NC-LAPACK routine `DGEEV` to test NC-LAPACK and its NC-BLACS dependency at the same time.

This generates an executable (`./gees`) that runs the eigenvalue solver. It performs a Schur decomposition by a call to `DGEEV` on a 14000-long square matrix.

You can see below an example of Makefile compiling `ncblacs.f90` that links against NC-LAPACK:

```

TARGETS = blacstest

FC = ifort
FFLAGS = -c -auto -qopenmp -fp-model source -ftz -O0 -ggdb -check bounds

CC = icc
CFLAGS = -c -O3
#CFLAGS = -c -O0 -gdb

BASE=/opt/
MKL=$(BASE)intel/mkl/

MKLINC = $(MKL)include/intel64/lp64
MKLINC95 = $(MKL)interfaces/include/intel64/lp64
MKLL = $(MKL)lib/intel64/
LIBSA = ./libscalapack_icc_rel.a $(BLACS) -L../nclapack -lnclapack
LIBMKL = -lmkl_core -lmkl_sequential -lmkl_intel_lp64
LIBS = -liomp5 -lm -lnuma ../ncblacs/ncblacs/libncblacs.so

```

```

OBJ = mol_par.o epsprofile.o discOrder4.o PDHSEQRTTEST.o ncblacs.o

.PHONY: all
all: $(TARGETS)

.PHONY: clean
clean:
    rm -f $(TARGETS) *.o

%.o: %.f90
    $(FC) $(FFLAGS) -I$(MPIINC95) -I$(MPIINC) -I$(MKLINC) $<

%.o: %.c
    $(CC) $(CFLAGS) $<

blacstest: $(OBJ)
    $(FC) -o gees -fopenmp $(OBJ) -I$(MKLINC) -L. -L$(MKL)lib/intel64 $(LIBSA)
    $(LIBMKL) $(LIBS)

```

/opt/ncblacs/EIG_TEST/Makefile_nclapack

Compiling the test *ncblacs.F90* into the executable *gees_NCLAPACK* **with NC-LAPACK**.

The user can check that the right libraries were used like this:

```

[nscale@node1 EIG_TEST]$ ldd ./gees_NCLAPACK
    linux-vdso.so.1 => (0x00007ffe1216f000)
    /opt/ncalloc/libncalloc.so (0x00007f881436d000)
    libnclapack.so => /opt/nclapack/libnclapack.so (0x00007f8814065000)
    libmkl_core.so =>
/opt/intel/compilers_and_libraries_2017.0.098/linux/mkl/lib/intel64/libmkl_core.s
o (0x00007f8812575000)
    libmkl_sequential.so =>
/opt/intel/compilers_and_libraries_2017.0.098/linux/mkl/lib/intel64/libmkl_sequen
tial.so (0x00007f88117b5000)
    libmkl_intel_lp64.so =>
/opt/intel/compilers_and_libraries_2017.0.098/linux/mkl/lib/intel64/libmkl_intel_
lp64.so (0x00007f8810c95000)
    libiomp5.so =>
/opt/intel/compilers_and_libraries_2017.0.098/linux/compiler/lib/intel64/libiomp5
.so (0x00007f881094d000)
    libm.so.6 => /lib64/libm.so.6 (0x00007f881062d000)
    libnuma.so.1 => /lib64/libnuma.so.1 (0x00007f881041d000)
    libncblacs.so => /opt/ncblacs/ncblacs/libncblacs.so (0x00007f88101f5000)
    libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f880ffd5000)
    libc.so.6 => /lib64/libc.so.6 (0x00007f880fc0d000)
    /lib64/ld-linux-x86-64.so.2 (0x00005578e5c95000)
    libgcc_s.so.1 => /opt/Numascale/lib64/libgcc_s.so.1 (0x00007f880f9f5000)
    libdl.so.2 => /lib64/libdl.so.2 (0x00007f880f7ed000)
    libimf.so =>
/opt/intel/compilers_and_libraries_2017.0.098/linux/compiler/lib/intel64/libimf.s
o (0x00007f880f2fd000)
    libsvml.so =>
/opt/intel/compilers_and_libraries_2017.0.098/linux/compiler/lib/intel64/libsvml.

```

```
so (0x00007f880e3e5000)
    libirng.so =>
/opt/intel/compilers_and_libraries_2017.0.098/linux/compiler/lib/intel64/libirng.
so (0x00007f880e05d000)
    libintlc.so.5 =>
/opt/intel/compilers_and_libraries_2017.0.098/linux/compiler/lib/intel64/libintlc
.so.5 (0x00007f880dded000)
    libncrc.so => /opt/ncrc/libncrc.so (0x00007f880dbcd000)
```

7.3.2 A.2.2 Solving an eigenvalue problem with NC-LAPACK

One interesting benchmark to run is also one of the most complex linear algebra problems: computing the eigenvalues and eigenvectors of a general matrix (DGEEV in LAPACK).

With such tests we can compare the performance of NC-LAPACK against one of the most used math libraries, MKL (Intel® Math Kernel Library).

The test shown below finds the eigenvalues and eigenvectors of a matrix of rank 14000. This NC-LAPACK calculation is running on 72 cores only and finishes the calculation in 620 seconds.

```
[root@node1 EIG_TEST]# pwd
/opt/ncblacs/EIG_TEST
[root@node1 EIG_TEST]# cat env.sh
export LD_PRELOAD=/opt/ncalloc/libncalloc.so
export MKL_NUM_THREADS=1
export OMP_NUM_THREADS=72
export BLACS_DBGLVL=3
export OMP_SCHEDULE=static
export OMP_STACKSIZE=64m
export OMP_WAIT_POLICY=active
export KMP_AFFINITY=granularity=fine,proclist=[0-143:2],explicit

[root@node1 EIG_TEST]# source env.sh
[root@node1 EIG_TEST]# ./gees
  DIMENSION    14000
  INIT A...
  LAPACK DGEEV...
  LWORK        1000
****INIT NCLAPACK::DGEEV, N = 14000...
lvectors 1, rvectors 0...
GRID 9 x 8, BLOCK 64 x 64
SCATTER 1 GB, 6.7 SEC, 222.4 MB/s
Reduce A to Hessenberg form...
Create Schur form...
totit 2393, sweeps 49, totns 24968
Calculate Eigenvectors...
GATHER 1.9 SEC, 786.7 MB/s
GATHER 1.9 SEC, 784.5 MB/s

  RUNTIME [SEC]    620.371708557941
```

```

424 -2.27337309477685      0.000000000000000E+000
425 -2.27330266535453      0.000000000000000E+000
436 -2.14019789521296      0.000000000000000E+000
437 -2.13925124320276      0.000000000000000E+000
448 -2.00726643760220      0.000000000000000E+000
449 -2.00258169203520      0.000000000000000E+000
450 -1.99465320533244      0.000000000000000E+000
451 -1.98822634420265      0.000000000000000E+000
MAIN DONE...
[root@node1 EIG_TEST]#

```

Large differences in performance should be expected between LAPACK and ScaLAPACK, simply because ScaLAPACK was built with scalability in mind. Likewise, similar difference in performance should be expected between MKL and NC-LAPACK, although MKL can exploit multi-threaded implementations.

When limited to 24 threads, MKL will solve the same problem in about 3784s (see below) that NC-LAPACK solved in 620s.

```

[root@node1 EIG_TEST]# pwd
/opt/ncblacs/EIG_TEST
[root@node1 EIG_TEST]# cat env.sh
export LD_PRELOAD=/opt/ncaloc/libncaloc.so
export MKL_NUM_THREADS=24
export OMP_NUM_THREADS=1
export BLACS_DBGLVL=3
export OMP_SCHEDULE=static
export OMP_STACKSIZE=64m
export OMP_WAIT_POLICY=active
export KMP_AFFINITY=granularity=fine,proclist=[0-23:1],explicit

[root@node1 EIG_TEST]# source env.sh
[root@node1 EIG_TEST]#

[root@node1 EIG_TEST]# ./gees > gees_MKL.out
DIMENSION    14000
INIT A...
LAPACK DGEEV...
LWORK        476000
[root@node1 EIG_TEST]# cat gees_MKL.out
DIMENSION    14000
INIT A...
LAPACK DGEEV...
LWORK        476000
[root@node1 EIG_TEST]# cat gees_MKL.out
DIMENSION    14000
INIT A...
LAPACK DGEEV...

```

```

LWORK          476000

RUNTIME [SEC]   3784.68518796843

    316  -2.27337309477685    0.000000000000000E+000
    317  -2.27330266535452    0.000000000000000E+000
    318  -2.14019789521258    0.000000000000000E+000
    319  -2.13925124320248    0.000000000000000E+000
    322  -2.00726643760199    0.000000000000000E+000
    323  -2.00258169203517    0.000000000000000E+000
    324  -1.99465320533233    0.000000000000000E+000
    325  -1.98822634420266    0.000000000000000E+000
MAIN DONE...
```

7.4 A.3 Numascope and vmxstat usage

To list the events supported by either tool:

```

$ vmxstat -list
events supported:
      nr_free_pages      unallocated pages
nr zone inactive anon    zone inactive anonymous pages
nr zone active anon      zone activate anonymous pages
nr zone inactive file    zone inactive file-backed pages
nr_zone_active_file      zone active file-backed pages
...
n2VicBlkXRecv            VicBlk and VicBlkClean commands received on Hypertransport
n2RdBlkXRecv             RdBlk and RdBlkS commands received on Hypertransport
n2RdBlkModRecv           RdBlkMod commands received on Hypertransport
n2ChangeToDirtyRecv      ChangeToDirty commands received on Hypertransport
n2RdSizedRecv            RdSized commands received on Hypertransport
```

To select events for monitoring:

```

$ vmxstat -events thp fault alloc,n2RdBlkXRecv,n2RdBlkModSent
thp fault alloc n2RdBlkXRecv n2RdBlkModSent
    0          1286          819
    0          1547          823
   99          1212          806
  953         1755203        110747
    0          618578        28154
    0          1812341       11963
    0          632112         679
 50042         557468       1027168
54870         572459        221342
 27745         489014       1082759
   190         28013        45256
```

For real time HTML5 graphing:

```
$ numascope
web interface available on port 80
```

SSH port forwarding may be used if the host's TCP port 80 isn't directly accessible. Additionally, the interface listening address may be specified via the `-listen` option. Finally, the default sample interval can be given in seconds as an optional argument, and if by default events should be reported *per-server* or averaged can be given by the `-discrete` option.

To illustrate the graphical interface, numascope was executed while the NASA Parallel Benchmarks (NPB) Lower Upper (LU) matrix transposition benchmark was running:



Here, with AMD HyperTransport Read Block commands being counted per-server, we can observe a highly imbalanced workload that stresses the interconnect from different servers at different times. The dynamic range is around a factor of 1000 (4500 to 4.5M events per second). This can be used by workload manager to deploy jobs elsewhere, programmers to balance their communication in a better way, use more local accesses or utilize caches more.

Kernel event counters are included to allow observing Virtual Memory activity that would stall application progress.

7.5 A.4 Questions and Answers

Question: Does NC-ALLOC help applications using mmap?

Answer: No, if an application uses only mmap it allocates directly from the OS not using the heap. Applications do this to allocate big chunks of memory. The size is multiples of 4KB pages, so that they cannot do this for every little string. Small pieces of memory should be allocated from the heap. The heap allocates the big blocks by using mmap and manages small blocks inside the big block.

Appendix B References

[ACTi_D1.1] D1.1: “ACTiCLOUD Requirements and base architecture”, ACTiCLOUD deliverable.

[ACTi_D1.2] D1.2: “ACTiCLOUD architecture”, ACTiCLOUD deliverable.

NC-ALLOC source code is available on <https://wiki.numascale.com>

Numascope and vmxstat source code is available at <https://github.com/numascale/numascope>